View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Wisconsin-Milwaukee

University of Wisconsin Milwaukee
UWM Digital Commons

Theses and Dissertations

December 2014

API Usage Verification Through Dataflow Analysis

Kenneth Baptiste Tapsoba
University of Wisconsin-Milwaukee

Follow this and additional works at: https://dc.uwm.edu/etd

 Part of the Computer Sciences Commons

Recommended Citation
Tapsoba, Kenneth Baptiste, "API Usage Verification Through Dataflow Analysis" (2014). *Theses and Dissertations*. 646.
https://dc.uwm.edu/etd/646

This Thesis is brought to you for free and open access by UWM Digital Commons. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of UWM Digital Commons. For more information, please contact open-access@uwm.edu.

API USAGE VERIFICATION THROUGH DATAFLOW ANALYSIS

by

Kenneth Tapsoba

A Thesis Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Master of Science

in Computer Science

at

The University of Wisconsin-Milwaukee

December 2014

ABSTRACT
API USAGE VERIFICATION THROUGH DATAFLOW ANALYSIS

by
Kenneth Tapsoba

The University of Wisconsin-Milwaukee, 2014
Under the Supervision of Professor Tian Zhao

Using APIs in a program is often difficult because of the incomplete documentation and the shortage of available examples. To cope with that, we have seen the increase of API checking tools that provide efficient suggestions for API usage. However, most of those checking tools use a pattern-based analysis to determine errors such as misuse of API calls. In this thesis, we introduce a different analysis technique that relies on explicit API state transitions for the analysis of the program. We adopt a static dataflow analysis framework from SOOT to inspect state transitions at each program point.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor Professor Tian Zhao for the support and guidance in helping me understand and model this thesis subject.

I would also like to recognize PhD student Ahmed Shatnawi for his help doing research and implementation of the product presented in this thesis.

Finally, I would like to thank all my family and friends who supported me throughout the completion of this project.

Chapter 1

Introduction

Application Programming Interfaces (APIs) specify how software components should interact with each other. They are small systems that provide services to other systems in order to execute an array of tasks. APIs range from small packages such as the CSV Reader to large ones such as the Java Application Programming Interfaces. They are usually referred as programming libraries or just APIs. Often, developers are faced with the lack of documentation or examples to properly use them. Hours are spent searching through the Internet or manuals, and numerous try-and-fails are done in order to understand the usage of an API, thereby leading to lengthy development phases. Testing is usually the preferred method of finding incorrect use of API within a program [1], but it is costly and time consuming. To cope with this issue, program analysis tools such as FindBugs [2], and CriticAL [3] have been modeled to ensure the proper use of APIs by developers. In fact, those tools represent for a programming language what the Spelling and Grammar tool represents for a word document. They detect improper use of the package within a program and they often provide a comprehensive usage guide. On one side, FindBugs looks through the Java classes in order to detect well-known bug patterns. It maintains a database of more than 200 bug patterns that is used as a reference during the analysis. CriticAL, one the other side, detects invalid API usage through a context-sensitive inter-procedural analysis. Using those tools, we were able to find misuse of API calls. However, some of the most apparent errors were missed.

1.1 Motivation

While using some of these API checking tools, we found that they were not able to properly detect the misuse or misplacement of API methods. For example, FindBugs was not able to detect some run-time errors such as making a database request after

closing the connection. By investigating the issue, we found that FindBugs conducts its analysis using a list of bug patterns that have been reported within its database. It is therefore not able to conduct an accurate analysis due to the fact that new bugs have not yet been entered into the database. By exploring other analysis tools such as PRIME [4], we noticed that it builds its API specifications through code snippets found on the Internet which limits its efficiency to its findings. Therefore, it is not able to build efficient specifications for seldom used libraries.

One simple approach to counter these restrictions would be to gather more API usage specifications from the Internet, and maintain an updated list of bug patterns in the database. Still, due to the fact that libraries have multiple usage rules, it will be time consuming to fetch and compile all of these rules for each API.

An alternative to searching the internet for examples or documentation is to build one's own documentation and examples using the method built by Buse and Weimer [5]. This method reads a program and generates complete API documentation and examples to using a Dataflow path sensitive analysis and a pattern abstraction technique. Nonetheless, this technique requires the program being read to use most if not all the API methods in order to build complete accurate specifications.

Another approach would be to use API specification mining tools [6] to capture the full extent of API specifications. However it requires a setting runtime environment and multiple test programs. Some mining tools or strategies are MAPO from Xie and Pei [7] and the Automata-Based Abstractions from Shoham et al. [8]. MAPO uses the signature of a method to query open source repositories to find the usage pattern of an API. The Automata-Based Abstractions approach developed by Shoham et al. mines API usage specifications within a Java class. However these techniques would have to be run on multiple set of programs to properly capture the entire scope of an API library.

Another way to keep an updated list of API specifications would be to track changes in API libraries as described by Uddin et al. [9]. But this approach has to be ran each time there is a new API library release.

Based on these findings, it not apparent how to properly define the usage rules of an API. Further tests of API checking tools reveal a limitation over the usage rules by the users. Indeed, the users have little room to add or modify API rules, which limits the flexibility of the analysis. FindBugs and CriticAL allow the user to add new or custom API usage rules, but then again, the user will have to manually implement specific classes which requires a technical knowledge of those tools. For example, creating a new API specification in CriticAL requires the implementation of 2 interfaces and 7 classes, and the extension of 2 classes. On top of doing the implementation of those classes and

methods, it is time consuming for the user. These findings inspired us to direct our research in finding a tool or an approach that capture the full scope of an API usage rules, while providing an easy way for the user to add new or custom rules. This tool should also be able to catch invalid API usage within the analyzed program.

In the next section we propose a solution that will provide an accurate program analysis while giving full control of the API specifications to the user.

1.2 Requirement

In finding a tool that can detect invalid API usage using an explicit definition of API rules, we first need a technique that eases the entry of new rules with sufficient information. For that our approach has to allow the user to define new API usage rules using a small number of implementations or extensions of core classes. Secondly, we would need to build a framework that uses these rules to verify proper usage of the API within a program. Our approach would have to inspect each line of code by determining its composition, API method calls, and variables affected by those method calls. To properly capture usage rules, we need to create a framework that gathers API specifications within a single instance as a dictionary. Those specifications will then be used throughout the program to check for invalid method calls. This framework should be flexible enough to allow the user to easily declare or alter API usage rules for the analysis. These two requirements define a starting point to our research to achieve the goal of our thesis.

1.3 Overview of Thesis

In this thesis, we first define program analysis and the components relevant to our thesis. We then use a case study in Chapter 3 as an example to construct a technique which is explained in Chapter 4. In Chapter 5, we present a prototype that implements our approach using the Java programming language. Experiments and results of the prototype are displayed in Chapter 6. Finally, we presents related works in Chapter 7, and our conclusion and future work in 8.

Chapter 2

What is Program Analysis?

2.1 Definition

According to the book Principles of Program Analysis by Nielson et al. [10],

> Program analysis offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviors arising dynamically at run-time when executing a program on a computer.

In other words, it is a way of inspecting a program for errors that commonly appear at run-time. It allows developers to check for infinite loops, null pointer exceptions, type cast exceptions, invalid API usage and more. The application of program analysis comes through different techniques of which the main ones are: Dataflow Analysis, Constraint Based Analysis, Abstract Interpretation, and Type and Effect Systems. Dataflow Analysis is a type of program analysis that inspects the stream of information available throughout the program. Constraint Based Analysis conducts an analysis by creating and solving a set of constraints produced by the program [11]. Abstract Interpretation is an analysis based on the semantics of the program. The Type and Effect System bases its analysis on the type annotations that describe intentional properties of the semantics of the program [12]. For the purpose of this thesis, we use Dataflow Analysis which is described further in the next section. An analysis can be done on a program by traversing it from the beginning of the program until the end, or from the end to the beginning. These are receptively called *forward-analysis* and *backward-analysis*. Throughout this thesis we will use the terms statement and expression that we define as follow:

- Statement (*Stmt*): A statement represents a line of code within a program. It is a series of commands that are executed when the program is running. In most

programming languages is it ended by a semicolon. Examples of statements are: $x = a + c * b$; or $a.m(y)$;. A statement can consist of an expression that has side effects such as a function call.

- Expression (*Expr*): An expression represents an operation that can be reduced to a value. $a + c * b$ or $a.m(y)$ are examples of expressions.

2.2 Dataflow Analysis

Dataflow Analysis, a commonly used program analysis technique, examines a program using the flow of data at each point of execution in the program. It determines how each component in the program interacts between each other [13]. In other words, it specifies the value of variables in the program and how they are used to impact the results of the computation. By doing so, Dataflow Analysis provides a set of possible outcomes of a program based on the input values. Dataflow Analysis uses a lattice and a set of transfer functions to execute its inspection of the program.

2.2.1 Lattice

By definition a lattice $L = (C, \sqsubseteq, \sqcup, \top, \bot)$ is a set of partially ordered objects in C such that if $x, y \in C$, there is a least upper bound ($\sqcup C$) and a greatest lower bound ($\sqcap C$) for $x \neq y$ [10]. It is a set of elements C arranged in a treelike structure with at least a common ancestor and a common child. $\bot$, called bottom, is the elements at the bottom and $\top$, called top, the elements at the top of the lattice. The combination operator $\sqcup$ determines how to combine information coming from different paths. The lattice's elements, ordered through a relation, composes a partial order $(C, \sqsubseteq)$ which represents an ordering and sequencing between elements of a given set through the relation $\sqsubseteq$ with the following properties:

- Reflexive: $\forall a \in C : a \sqsubseteq a$
- Anti-symmetric: $\forall a, b \in C : a \sqsubseteq b \wedge b \sqsubseteq a \rightarrow a = b$
- Transitive: $\forall a, b, c \in C : a \sqsubseteq b \wedge b \sqsubseteq c \rightarrow a \sqsubseteq c$

It shows the ordering conservation and the reflexive behavior of the lattice. Figure 2.1 shows a representation of a lattice $L = (C, \sqsubseteq, \sqcup, \top, \bot)$ where the set of elements $C = x, y, z$, the top of the lattice $\top = x, y, z$ and the bottom $\bot = \emptyset$. The order relation is

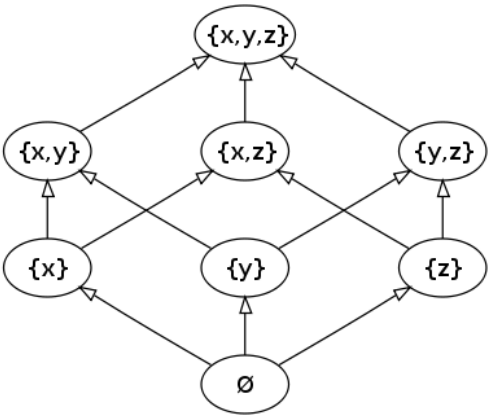


FIGURE 2.1: Lattice

$\sqsubseteq = \subseteq$, and the combination operator is $\sqcup = \cup$ because we notice a union of elements as we move from the bottom of the lattice until the top.

The lattice is used within Dataflow Analysis to determine the value of the variables within the program. It presents a hierarchy between values which allows us to direct flow through variables within the program on statements such as if-statements, loops, switch, etc.

2.2.2 Transfer Function

At each statement there are chances that the abstract value of one or multiple variables gets updated. Each update is captured by a function which determines how to compute the new value of the variable. The set of those functions is call transfer functions $\mathcal{F}$. They are the set of updates for each statement in the program. We denote:

$$f_l : L \to L$$

This shows that the domain of a transfer function is within the lattice itself [10]. In other words, it takes elements from a lattice and outputs new elements that are in the same lattice. In order to find the abstract value of a variable at an expression we use the formula $A[\![x]\!]\sigma = \sigma(x)$ where x is a variable in the expression. Dataflow analysis uses transfer functions to find an approximation of the path taken by a program at run-time.

2.2.3 Monotone Framework

A monotone framework is composed of a complete lattice L and a set of monotone transfer functions $\mathcal{F}$ [10, 14]. A complete lattice is a lattice with a unique lower upper

bound and a unique greatest lower bound for any two elements. A transfer function f is considered monotone only if $a \sqsubseteq b \rightarrow f(a) \sqsubseteq f(b)$. It shows the conservation of the ordering of a and b after applying the transfer function. We use this characteristic of the program analysis in our approach to show the order conservation of the lattice.

2.3 Features of Program Analysis

In order to conduct an accurate analysis, a program analyst might want to specify some feature of the analysis such as intra-procedural vs. inter-procedural, and context-sensitive vs. context-insensitive.

2.3.1 Intra-Procedural vs. Inter-Procedural Analysis

An intra-procedural analysis is considered as one that only examines statements within a function while an inter-procedural analysis allows us to examine statements inside method referenced within a method. Let's consider the code snippet and the program flow graphs in Figure 2.2.

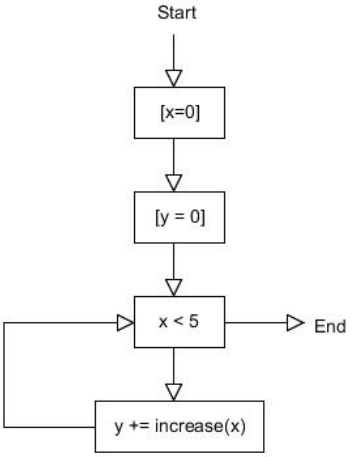
We see that the flow graph of the intra-procedural analysis is maintained within the contents of the main method while the flow graph of the intra- procedural analysis is composed of the contents of the method increase being referenced in the method main. Truly, an inter-procedural analysis is an intra-procedural analysis with the addition of edges leading to the source of methods being referenced. For that, the intra-procedural analysis crosses the boundaries of methods and therefore allows for the evaluation of the contents of remote methods. It increases the accuracy of the analysis. In our approach, we will combine both concepts such that an intra-procedural analysis will be conducted within a method, and an inter-procedural analysis will be used when a remote method is called within the currently analyzed method.

2.3.2 Context-Insensitive vs. Context-Sensitive Analysis

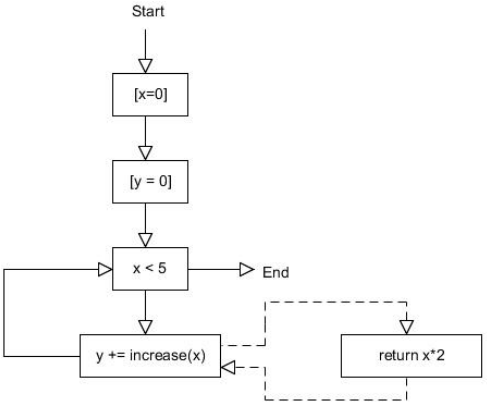
While a context-insensitive analysis considers a method call the same at all the places the method is called, a context-sensitive analysis considers each method call differently. In other words, if two calls are made to the method *printResults()*, the context-sensitive analysis treats them differently because the value of the variables are likely to be *different*. In some programs, the context sensitivity is implemented by using a call string. The call string is a reference that records the past 1 to k sites leading to the current method call. In general it is good practice to use the label determined for each line

```
public class Main {

      public static void main(String[] args){
            int x = 0;
            int y = 0;
            while (x < 5) {
                  y += increase(x);
            }

      }

      public static int increase(int x){
            return x*2;
      }
}
```

(A) Code snippet



(B) Intra-Procedural



(C) A Inter-Procedural

FIGURE 2.2: Intra-Procedural vs Inter-Procedural

of code of the program [10]. Let's consider the example, in Figure 2.3. By having a context-sensitive analysis, the variable $c21$ will have a status of connected while the variable $c11$ will not be connected after the computation. This is due to the fact the function f being called is considered to be in different contexts for line 5 and line 6. Using the call string, the first call to function f will have call string of 3 and the second call with have call string of 4, respectively referred by label 3 and label 4 in the code. Therefore, the call $f(c1)$ is treated differently from the call $f(c2)$, which allows us to determine that variable $c21$ is still connected while variable $c11$ is not connected. In case the analysis is context-insensitive, the statuses of $c11$ and $c21$ will both be closed at the end of the computation. This is because the change of status by $c11$ will get propagated to all the locations in the program making the same call such as the variable $c21$. The status of $c21$ will then be updated to the status of not connected. In our approach, we use the context-sensitivity technique to accurately determine the status of each variable

```
1
2  foo(){
3  c1 = getConnection();    //label 1
4  c2 = getConnection();    //label 2
5  c11 = f(c1);             //label 3
6  c21 = f(c2);             //label 4
7  c11.close();             //label 5
8  }
9
10 //identity function
11 f(x) => x;
12
```

FIGURE 2.3: Context-Insensitive vs. Context-Sensitive

at each execution point in the program. This will allow us to properly detect invalid API calls.

Chapter 3

Case Study: Java Database Connectivity Library

This section illustrates the study of the Java Database Connectivity (JDBC) library during a code analysis. We use the code in Figure 3.1 as a sample Java program that connects, queries, prints results from a database, and closes the connection.

From it we see that the JDBC library is transitioning through a set of stages by calling the methods: `getConnection()`,`createStatement()`,`executeQuery()`, and `close()`. Considering that each method alters the state of the library, the graph in Figure 3.2 is constructed showing the stages as each of the methods are called.

From Figure 3.2, we could determine a set of stages traversed by the library:

NotConnected : No connection has been established with the database. The method used to reach this stage is `Connection.close()`;

Connected : A connection has been established with the database. The method used to reach this stage is `DriverManager.getConnection()`;

Statement : An execution statement has been created in order to request data from the database. This stage is reached by the method `Connection.createStatement()`;

StatementClosed : This stage shows that the execution statement has been removed. It is reached by the method `Statement.close()`;

ResultSet : This stage depicts the aftermath of executing the statement defined earlier. It might retrieve some data from the database. It is reached through the method `Statement.executeQuery()`;

```
1  public static void main(String[] args) {
2      Connection conn;
3      try{
4          Class.forName("com.mysql.jdbc.Driver");
5          conn = DriverManager.getConnection(DB_URL,USER,PASS);
6          Statement stmt = conn.createStatement();
7          ResultSet rs = stmt.executeQuery("SELECT * from country");
8
9          while(rs.next()){
10             ...
11         }
12         rs.close();
13         stmt.close();
14         conn.close();
15
16     }catch(Exception e){
17         //Handle errors for Class.forName
18         e.printStackTrace();
19
20     }finally{
21         try{
22             if(conn!=null)
23                 System.out.println("Close Connection again");
24                 conn.close();
25
26         }catch(SQLException se){
27             se.printStackTrace();
28
29         }//end finally try
30
31     }//end finally
32
33 }//end main
```

FIGURE 3.1: Case Study code example

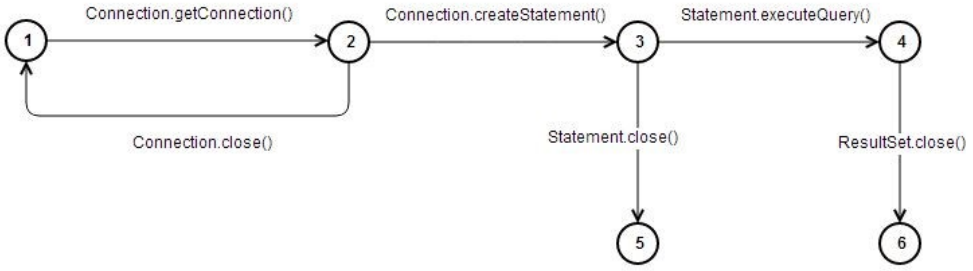


FIGURE 3.2: Stages of JDBC API

Stages/Phases	Valid Methods
NotConnected	`DriverManager.getConnection()`, `Connection.close()`
Connected	`DriverManager.getConnection()`, `Connection.createStatement()}`, `Connection.close()`
Statement	`Connection.createStatement()`, `Statement.executeStatement()`, `Statement.close()`
StatementClosed	`Statement.close()`
ResultSet	`Statement.executeQuery()`, `ResultSet.next()`, `ResultSet.getAsString()`, `ResultSet.close()`
ResultSetClosed	`ResultSet.close()`

TABLE 3.1: Valid Methods

ResultSetClosed : This stage shows that the ResultSet stage has been removed and the data that it contains has been deleted. The method used to reach this stage is `ResultSet.close()`.

From there, a Dataflow analysis can be done to records different changes in the state of the library. A common run-time error of the JDBC library is to request data from a database without having a connection. A good practice is that a person's code should first create a connection before requesting data from a database. We can deduct that there is an order of stages while using the JDBC library. The method `getConnection()` is always called before the other methods described earlier. We are then able to build a set of valid methods for each stage shown in table 3.1.

We notice that each phase is reached by an explicit method call and our example followed those rules. There are no errors or invalid calls obtained after the analysis. Now let's consider the example of Figure 3.3, where the connection is closed before creating a statement.

At run-time, this example throws an error because it is not possible to create a statement with a closed connection. API checking tools like PRIME [4] are able to capture the error. It uses a pattern analysis to detect that the sequence of methods `close()` and `createStatement()` is not a valid one. Another checking tool such as FindBugs [2], does not detect the error because it does not consider the different stages of the API. Plus, adding a new API usage rule to FindBugs has been found to be complicated [15]. Based on those findings, we strive to design an approach that will verify API usage and users' input to detect invalid API method calls.

```
1  public static void main(String[] args) {
2      Connection conn;
3      try{
4          Class.forName("com.mysql.jdbc.Driver");
5          conn = DriverManager.getConnection(DB_URL,USER,PASS);
6          conn.close();
7          Statement stmt = conn.createStatement();
8          ResultSet rs = stmt.executeQuery("SELECT * from country");
9
10         while(rs.next()){
11             ...
12         }
13         rs.close();
14         stmt.close();
15         conn.close();
16
17     }catch(Exception e){
18         //Handle errors for Class.forName
19         e.printStackTrace();
20
21     }finally{
22         try{
23             if(conn!=null)
24                 System.out.println("Close Connection again");
25                 conn.close();
26
27         }catch(SQLException se){
28             se.printStackTrace();
29
30         }//end finally try
31
32     }//end finally
33
34 }//end main
```

FIGURE 3.3: JDBC code example with error

Chapter 4

Approach

Recalling the requirements of our approach being to properly capture API specifications and verify their proper usage, we constructed an approach using a set of configurations and the Dataflow Analysis framework. The set of configurations constitute the different API usage rules defined by the user and used by Dataflow Analysis to find invalid API calls. These two components constitute the core of our analysis, and we elaborate more on them in the following lines.

4.1 Configuration

4.1.1 Structure

A configuration could be described as the place having all the rules concerning the use of an API. It is a dictionary that contains API methods and a list of interactions between those methods. In this paper, we distinguish a method as a function that execute a specific task. It is composed of its name, and the name of the class where it is defined. We use the notation `M` to represent a method. For example, the method `Connection.getConnection()` has for name `getConnection` and for class name `Connection`. Calling a method from an API library might alter the status of that library. We call that an action, represented by the symbol `A`. Because an action can be triggered by one or several methods we specify it with a list of methods. Here the action for calling the method `connection.getConnection()` has the list of methods {`Connection.getConnection()`}. To represent the status of an API at any given moment during the analysis, we use state `S`. A state is reached through the application of one or several actions. For that a state is composed of a name and a list of actions. We have the state `Connected` with the action {`Connection.getConnection`}. When

an API moves from one state to another using an action, we call it a Transition. A transition represents the change of state of an API going from `stateA` to `stateB` using `actionX`. As an example, we can consider the API change from the state `NotConnected` to the state `Connected` using the action {`Connection.getConnection`}. With these definitions, we can specify each one using:

$$\begin{aligned} M &::= \langle name, class \rangle \\ A &::= \langle M_1, M_2, \ldots, M_n \rangle \\ S &::= \langle A_1, A_2, \ldots, A_n \rangle \\ T &::= \langle S_{start}, S_{end}, \ldots, A_n \rangle \end{aligned}$$

The interaction between the set of methods, actions, states, and Transitions allows us to properly capture the behavior of an API library. By setting those, we are able to specify how an API evolves through different stages by its method calls. Those sets constitute the configuration C of an API described as:

$$C ::= \langle \{M\}, \{A\}, \{S\}, \{T\} \rangle$$

In addition of maintaining API rules, it is possible to retrieve each component of a configuration using the following equations:

- $M_{\langle n,c \rangle} ::= C[n, c]$: returns the method with the name n and the class c.
- $\{A_{\langle M_1, M_2, \ldots, M_n \rangle}\} ::= C[M_x]$ with $x \in 1..n$: returns the set of actions associated with the method M_x.
- $\{S_{\langle A_1, A_2, \ldots, A_n \rangle}\} ::= C[A_x]$ with $x \in 1..n$: returns the set of states associated with the action A_x.

Equipped with these models, we are able to capture the full scope of API specifications defined by the user.

4.1.2 Example: JDBC Library

Using the characteristics of the JDBC library from Chapter 3, we built the following definitions for methods, actions, states, and Transitions. Because of the extensive number of methods in the JDBC library, we considered those that are likely to change the state of the library. We also represent a method by using only its name because of its uniqueness. For example the method `Connection.getConnection()` is represented by simply `getConnection`.

Methods

$$
\begin{aligned}
M_{getConnection} &::= < getConnection, DriverManager > \\
M_{createStatement} &::= < createStatement, Connection > \\
M_{executeQuery} &::= < executeQuery, Statement > \\
M_{closeResultSet} &::= < close, ResultSet > \\
M_{closeStatement} &::= < close, Statement > \\
M_{closeConnection} &::= < close, Connection >
\end{aligned}
$$

Actions

$$
\begin{aligned}
A_{getConnection} &::= < \{M_{getConnection}\} > \\
A_{createStatement} &::= < \{M_{createStatement}\} > \\
A_{executeQuery} &::= < \{M_{executeQuery}\} > \\
A_{closeResultSet} &::= < \{M_{closeResultSet}\} > \\
A_{closeStatement} &::= < \{M_{closeStatement}\} > \\
A_{closeConnection} &::= < \{M_{closeConnection}\} >
\end{aligned}
$$

States

$$
\begin{aligned}
S_{connected} &::= < \{A_{getConnection}\} > \\
S_{statement} &::= < \{A_{createStatement}\} > \\
S_{resultSet} &::= < \{A_{executeQuery}\} > \\
S_{resultSetClosed} &::= < \{A_{closeResultSet}\} > \\
S_{statementClosed} &::= < \{A_{closeStatement}\} > \\
S_{notConnected} &::= < \{A_{closeConnection}\} >
\end{aligned}
$$

Transition

$$
\begin{aligned}
T_{toConnected} &::= < S_{notConnected}, S_{connected}, A_{getConnection} > \\
T_{toStatement} &::= < S_{connected}, S_{statement}, A_{createStatement} > \\
T_{toResultSet} &::= < S_{statement}, S_{resultSet}, A_{executeQuery} > \\
T_{toResultSetClosed} &::= < S_{resultSet}, S_{resultSetClosed}, A_{closeResultSet} > \\
T_{toStatementClosed} &::= < S_{statement}, S_{statementClosed}, A_{closeStatement} > \\
T_{toNotConnected} &::= < S_{Connected}, S_{notConnected}, A_{closeConnection} > \\
T_{toConnected} &::= < S_{connected}, S_{connected}, A_{getConnection} > \\
T_{toStatement} &::= < S_{statment}, S_{statement}, A_{createStatement} > \\
T_{toResultSet} &::= < S_{resultSet}, S_{resultSet}, A_{executeQuery} > \\
T_{toResultSetClosed} &::= < S_{resultSetClosed}, S_{resultSetClosed}, A_{closeResultSet} > \\
T_{toStatementClosed} &::= < S_{statementClosed}, S_{statementClosed}, A_{closeStatement} > \\
T_{toNotConnected} &::= < S_{notConnected}, S_{notConnected}, A_{closeConnection} >
\end{aligned}
$$

We notice that the last 6 transitions have the same starting and ending state. This is explained by the fact that the API method is called without modifying its state. For example, calling multiple times the method `getConnection()` re-initializes the connection to the database but the API state is still in the Connected state. It shows that a transition can have the same starting state and ending state after applying an action. With the API rules defined, we can now use the Dataflow Analysis framework to detect invalid API calls within a program.

4.2 Dataflow Analysis

Dataflow analysis is a program analysis technique that determines the value of variables throughout the program. In our case, we use this technique to approximate the set of states for variables throughout the program. In other words, variables are commonly used to initiate or store contents from API calls. By focusing on the program variables, we are able to get an approximation of the state of the API currently used within the program. In addition, we will use a monotone framework $(L, \mathcal{F})$ within our instance of Dataflow analysis to show the order conservation of elements within the lattice. The monotone framework will also provide a base to explore and evaluate each line of code using the appropriate transfer function. The analysis will generate a map containing a set of states for variables before and after the examination of a statement. In the next sections, we depict the stages of this analysis and how we adapted their use to detect invalid transitions.

4.2.1 Lattice

We build our lattice, $L = (C, \sqsubseteq, \sqcup, \top, \bot)$, on a set of partially ordered states that maps a variable to a subset of states at each statement in the code. The part $(C, \sqsubseteq)$ represents the partial order of states where C is the set of states ordered through the relation $\sqsubseteq$. Because at each statement, a variable can be mapped to a state, we therefore use a power set of states $\mathcal{P}(States)$ for the lattice. By using a forward analysis we assume that all the variables do not have a state and they progressively evolve through the lattice toward a top at the beginning of the analysis. Therefore the bottom state $\bot$ is the empty state $\emptyset$, and the top state $\top$ is the set of all states $\{States\}$. It also means that the set of states at the entry of a statement is inclusive to the set of states at the exit as a variable moves from the empty state to the top state. For that, the order relation $\sqsubseteq$ is $\subseteq$. In our analysis, we are exploring all the possible states of a variable at each stage of execution. In other words, for a starting state and an action, we consider all the valid output states that a variable might end up with. It means that our analysis is a *May-analysis* and we are interested in finding the greatest set of states that satisfy the conditions of the transition. Thus, the combination operator $\sqcup$ is $\cup$. As a result our lattice is:

$$L = (\mathcal{P}(States), \subseteq, \cup, \{States\}, \emptyset)$$

Figure 4.1 represents the JDBC library lattice built based on our case study. We see that it starts from the empty set to one state, then it evolves through a series of combination with different states (represented by the dashed lines) to a set of all the states, the top state. The subsets $\{NotConnected\}$, $\{NotConnected, Connected\}$, and $\{NotConnected, Connected, Statement\}$ are intermediate nodes between the empty state and the top state.

4.2.2 Transfer Functions

Our approach uses transfer functions to compute a set of output states for each variables at each statement. The set of states for a variable x is obtained through $\sigma(x)$ and the set of states of x at an expression is $A[\![x]\!]\sigma = \sigma(x)$. During the evaluation of each statement, we concentrate on two types of statements: assignment statements and invoke statements. Assignment statements are statements of the kind $[x = e]$ where variable e is an expression and x is a variable. Invoke statements are of the kind $x.m(y)$ where m is a method and y is the list of arguments passed to the method. With those two statements we are able to capture any API calls within the program. Other statements such as conditional statements or return statements are composed of either assignment statement or invoke statements or just a regular expression. This is why, we focus on

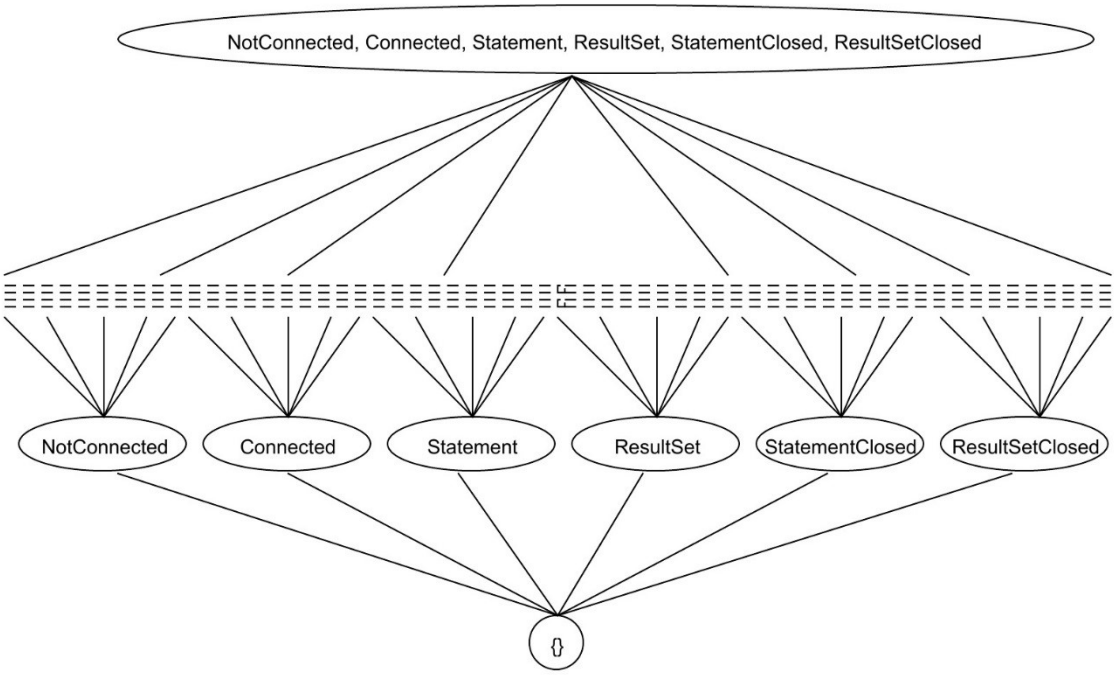


FIGURE 4.1: JDBC Lattice example

assignment statement and invoke statements. We evaluate an assignment statement and an invoke statement using the following equations:

(1) $f[x = e]\sigma = \sigma[x \rightarrow A[\![e]\!]\sigma$

(2) $f[x.m(y)]\sigma = \sigma[x \rightarrow S]$ where $S = \{S_{end} \in T[S_{start}, S_{end}, A_x]\}$ for each $S_{start} \in A[\![x]\!]\sigma$ and $m \in \{M_1, M_2, \ldots, M_n\}$ in A_x

The evaluation of the assignment statement in equation (1) shows that the output states of variable x comes from the states of the expression e. In other words the variable x will be mapped to the set of states of the expression e. In the case where the expression e is a variable, its corresponding sets of states is returned. If e is a method call, it follows the same path as the evaluation of the invoke statement. The equation (2) that evaluates the invoke statements shows that x is mapped to the set of states returned from the valid transitions using the states of x as starting states and the action a triggered by the method m. In other words, the result of the equation is the set of end states for each valid transition composed by a state in $A[\![x]\!]\sigma$ and an action A_x. This equation looks for valid transitions for the states of x. In case there are no valid transitions, x does not have the prerequisites to call method m which means that it is an invalid transition. In case one or more transitions are found, their end states S_{end} are collected and assigned to x. This is how we define and evaluate our transfer functions used by

Dataflow analysis to compute abstract states of variables for each statements, and to find invalid state transitions.

4.3 State Dependency

Our approach introduces the concept of state dependency in the analysis. This is important because a state change of a variable might cause a state change in other variables. This is illustrated when a statement is executed after closing its connection to the database. We found that a statement variable using the same connection as the connection variable can no longer be executed if the connection variable closes the connection. This fact makes the statement variable dependent on the connection. However, if the connection variable creates a new connection afterward, the statement variable is no longer dependent on the connection variable because they are using different connections. To capture this we create a relationship between the parent and the dependent variable by using a map that takes the parent as a key and a list of dependents as a value. We also define a base state for each state. The base state of a variable is a state that when reached, it triggers all the variable's dependencies to be updated to their base states. In the JDBC example, the base state for *Connected* is *NotConnected*. When a variable x transitions from *Connected* to *NotConnected*, all the dependents of x will automatically transition to their base states. The base state for *Statement* is *StatementClosed*, and it is *ResultSetClosed* for *ResultSet*. For then if a connection is closed, all the statements relying on that connection will also be closed. Also, all the result sets relying on those closed statements will be closed. In case variable x gets a new connection, there is no more dependency relationship between x an all the statements that were using its connection. This will remove any previous dependencies on the connection.

Chapter 5

Implementation

In order to put our approach to test, we have implemented a prototype that formalizes the concepts explained in the previous sections. We adapted an open source framework called SOOT to conduct the analysis. SOOT provides a framework to do a flow sensitive analysis through the transformation of the Java code into a Java byte-code [16]. This transformation allows for the manipulation of each statement within a Java program. We also modified Dataflow analysis technique from VASCO [17] to allows us to determine the set of states of each variable throughout the analysis. Our implementation is structured in 3 main packages: (1) the Resource package that represents the core of our approach; (2) the Program Analysis package which conducts a full program analysis; (3) and the Verification package that evaluates transfer functions and detects invalid API calls. In the following lines we will describe SOOT and how it is being used, then we will explain each of the main packages used within the implementation and how they interact between each other.

5.1 SOOT Analysis Framework

SOOT, initially released in 1999, is a program analysis framework developed by the Sable Research Group at McGill University, Montreal, Canada. It provides a blueprint to conduct static program analysis by transforming the Java code into intermediate representations such as Baf, Jimple, Shimple and Grimp [18]. The Baf transformation is a ”streamlined stack-based representation of byte-code” that provides an abstraction of the Java byte-code as a stack code. The Jimple transformation is a three address representation of the Java byte-code. It dissembles Java code into simple and easy instructions having 1 operation and 2 operands - see Figure 5.1. The Shimple transformation is similar to the Jimple transformation at the exception that each variable is

```
public class Foo {

  public static void main(String[] args) {
    Foo f = new Foo();
    int a = 7;
    int b = 14;
    int x = (f.bar(21) + a) * b;
  }

  public int bar(int n) { return n + 42; }
}
```

(A) Java code

```
public static void main(java.lang.String[]) {
  java.lang.String[] r0;
  Foo $r1, r2;
  int i0, i1, i2, $i3, $i4;

  r0 := @parameter0: java.lang.String[];
  $r1 = new Foo;
  specialinvoke $r1.<Foo: void <init>()>();
  r2 = $r1;
  i0 = 7;
  i1 = 14;
  // InvokeStmt
  $i3 = virtualinvoke r2.<Foo: int bar()>(21);
  $i4 = $i3 + i0;
  i2 = $i4 * i1;
  return;
}
```

(B) Jimple representation

FIGURE 5.1: Java code to Jimple 3 address representation

considered to have a static point of definition. The Grimp transformation is also similar to the Jimple transformation except that its representation is similar to the Java code. In our experiment, we decided to use the Jimple transformation because it decomposes the Java byte-code into a 3 address instruction. This allows us to optimize our analysis by focusing on what is happening at each steps of the statement instead of how the statement is composed.

In Figure 5.1 [18], we notice that the statement `int x = (f.bar(21) + a) * b;` in the Java code is broken down into several statements in the Jimple transformation. This breaks down a group of actions into multiple single steps, making each statements easily identifiable. In addition to providing those transformations, SOOT provides a framework for Dataflow analysis and Points-To analysis. In our case, we are using the Dataflow analysis framework by extending one of its classes: `ForwardFlowAnalysis`.

The `ForwardFlowAnalysis` analyzes the program from the first statement to be executed in the program until the last, hence the name Forward Flow. Furthermore, we use SOOT to make our analysis inter-procedural though the use of the `callgraph`. In fact, the `callgraph` is a class containing all the classes, and methods of the program being analyzed. It is used as a reference to find available methods and functions. Another feature of SOOT is the context-sensitivity which can be turned on or off for the analysis. In our case, we turn on the context-sensitivity to differentiate contexts when a method is called multiple times at different locations in the code. With these features, SOOT gives us a blueprint to implement our analysis.

5.2 The Resource Package

The resource package is composed of the core components of our analysis. It has the `Method` class, the `Action` class, the `State` class, and the `Transition` class that represent respectively a method, an action, a state, and a transition described from the approach.

5.3 The Program Analysis Package

The Program Analysis package contains all the classes needed to explore each statement of the program. In this package we have the classes `Analysis`, `ControlFlowGraph`, `DataflowAnalysis`, and `StatementVisitor`.

5.3.1 The Analysis

The `Analysis` class is where the transformation of the Java byte-code takes place by extending a `BodyTransfomer` provided by SOOT. In this class we specify the type of representation needed (Jimple in our case) and SOOT takes care of the rest. The `Analysis` class serves as a starting point for our analysis. When running, this class transforms the Java byte-code into a Jimple representation. It also initializes an instance of the `ControlFlowGraph`. It then starts the analysis by passing the instance of the `ControlFlowGraph`, and the Jimple representation to the `DataFlowAnalysis`. This `Analysis` also serves as a place for the user to define API usage rules.

5.3.2 The ControlFlowGraph

The `ControlFlowGraph` is used to store the `callgraph`. It provides the analysis with methods that can determine if a function call is intra-procedural or inter-procedural. In case the call is inter-procedural, the `ControlFlowGraph` uses the call-graph to provide the contents of the method called. That allows us to conduct an inter-procedural analysis. Table 5.1 shows the functions of the `ControlFlowGraph` class that we used in our analysis [19]:

5.3.3 The DataflowAnalysis

Dataflow Analysis represented by the DataFlowAnalysis class is where our analysis takes place. In other words, it explores each statement and determines the states of variables before and after the execution of a statement. To navigate through each statement, this

Method Signature	Method Description
`InterProceduralCFG()`	Constructor. It gets the methods and bodies of the program from the Callgraph
`isBranchStmt(stmt)`	Returns true if the contents of a method are available to be explored
`isCallStmt(stmt)`	Returns true if it is an API method call
`getArguments(stmt)`	Returns the list of arguments that are being passed to a method
`getParameters(stmt)`	Returns the list of parameters of a method
`makeGraph(stmt)`	Creates a Control Flow Graph from the body of a method
`getLocals(stmt)`	Returns the list of local variables of a method

TABLE 5.1: Control Flow Graph Method

class is extended to the `ForwardFlowAnalysis` class. For that, we had to specify the following parameters to determine the behavior of the analysis :

The graph : The graph represents $flow(S^*)$ the flow of the program being analyzed. It is a set of linked nodes that shows how the statements of the program are connected with each other. The graph is built by the Jimple Transformation in the Analysis class and passed to the `DataflowAnalysis` to navigate through the statements.

Type of analysis : In our case, the type of analysis is `State` because we are interested in the states of each variables making an API call

Type of the objects being analyzed : it will be the type `Local` of variables present in the program because local variables hold the state of API libraries through API calls or assignments. The type Local comes from the Jimple transformation to recognize variables.

EntryInitialFlow(ι) : It represents the state of the program before executing a statement. Since we are conducting a forward analysis, the `EntryInitialFlow` will be the exit flow of the statement previously executed at the exception of the first statement where the `EntryInitialFlow` will be empty: $\iota = \emptyset$.

NewInitialFlow : It represents the default state of the program before the execution of a statement. Because we consider that a program could start from any statement, the `NewInitialFlow` is empty.

Merge : It merges two input flows into one output flow. This happens when a statement's reachability is determined by a condition. We often find these cases in loop statements, switch statements, or if statements. In case, the input flows have the same variable mapped to different states, we use the formula of the least upper bound to find the output flow of this variable.

Copy : It copies the contents of `flowA` into `flowB`. It is used to copy the state of the program after executing a statement to the state of the program before executing the next statement. This function maintains the state of the program transitioning from one statement to the next.

FlowThrough : This is where the bulk of the work is done. The `FlowThrough` function allows us to explore the expressions of each statements and determine the set of transfer functions to apply for each variables. This is where we check for API calls, state transitions, and invalid API calls. This method uses the graph to determine the execution paths.

To navigate through each statement, SOOT maintains a method called `doAnalysis()` which determines which statement to analyze next. The method `doAnalysis()` uses the functions described above to compute the state of the program at any given point in time. The function `FlowThrough` explores each statement using the `StatementVisitor` class that determine the type of transfer functions to create.

5.3.4 The StatementVisitor

The `StatementVisitor` traverses each statement by depicting the operations that it executes. It explores each expression and creates the appropriate set of transfer functions for each of the variables whose value is being updated. It also uses the `ControlFlowGraph` to detect inter-procedural method calls. If any, the `StatementVisitor` conducts the following tasks:

1. Match the parameters to the arguments of the method called.
2. Use the `ControlFlowGraph` to get the graph of the function being called.
3. Create a new instance of the `DataflowAnalysis` using the newly created graph, the state of the program, the `ControlFlowGraph`, and the `Configuration`.
4. Run the analysis for the newly created `DataflowAnalysis`.
5. When the analysis is done, match back the states of the parameters to their resulting states after the analysis.
6. Remove any variables that exists within the boundaries of the method called.
7. If the method returns an object, assign the state of the object to the appropriate variable.

For an API call, we use the equations from section 4.2.2 to determine the set of transfer functions for each statement. In short, the Analysis Package provides the necessary classes to explore each method, statement and class that exits within a program. This package make use of the Resource Package to annotate the code, and it also uses the Verification package to determine valid and invalid API calls.

5.4 The Verification Package

In this package, we have included the classes used to check valid API calls. It implements the `transferFunction`, the `Configuration`, and the `InvalidTransitionException` classes.

5.4.1 The TransferFunction

The `TransferFunction` builds the equation used to compute the states of a variable based on its previous value. In other words, it is the representation of the Transfer function described earlier in the Approach section. It contains the following methods:

TransferFunction : the constructor of the class. It takes a starting state and the statement being analyzed.

Apply : It computes the output states after executing the statement. In other words, this method verifies transitions using the starting state, a set of output states and an action. The set of output states is a list of states reachable using the action called at the statement. The apply method looks for a valid transition between the starting state and each of the output states. If no valid transition is found, an error is thrown using the `InvalidTransitionException` class. Otherwise, a new list of states is returned. This list of states represents the states reachable from the starting state using the action. The verification of a valid transition is done by calling the method `checkTransition()` from the `Configuration`.

5.4.2 The Configuration

This class is the implementation of the configuration model from Chapter 4. It contains the API usage rules defined by the user. It also serves as an API usage checking tool to determine valid or invalid API calls through its method *checkTransition()*. It uses two criteria to determine the validity of a transaction:

The existence of the transition : In this case, we check if the transition (composed by the starting state, the ending state and the action) has been defined by the user. If it has been defined, the method returns true, and false otherwise.

Validity of the starting state : In this case we check for a transition using a null State. It would mean that a variable with a null value is making an API call. We consider it illegal because a null pointer exception is thrown when the program runs. Therefore we throw an error using the `InvalidTransitionException` class. By doing so, we are able to prevent null pointer exceptions on the usage of API libraries.

Because this class uses a set of `Methods`, `Actions`, `States` and `Transitions`, it contains easy-to-use methods shown below to add those sets. Those methods are called within the `Analysis` class to define each method, action, state, and transition for the API analyzed.

Add new Method :

`addNewMethod(name, class)`

Example: `addNewMethod(Connection, createStatement)` for the method $M_{createStatement}$

Adding new Action :

`addNewAction(name, methodName)`

Example: `addNewAction(getStatement, Connection.createStatement)` for the action $A_{createStatement}$

Adding new State :

`addNewState(name,orderIndex, baseStateName)`

Example: `addNewState(Statement,2,StatementClosed)` for the state $S_{statement}$

Adding new Transition :

`addNewTransition(startStateName, endStateName, actionName)`

Example: `addNewTransition(Connection, Statement, createStatement)` for going from $S_{connection}$ to $S_{statement}$ using $A_{createStatement}$

5.4.3 The Invalid Transition Exception

Once an invalid transition is found, an error is thrown using the `InvalidTransitionException` class. This class extends the `RuntimeException` class that catches exception in the program. When an invalid transaction exception is raised, a custom message is created using the starting state used, the action called, the variable involved, and the statement

```
Error!! Invalid API call.
Statement temp$5 = interfaceinvoke conn.<java.sql.Connection:
java.sql.Statement createStatement()>()
Starting state: NotConnected
action: Connection.createStatement()
variable: con
```

FIGURE 5.2: Example of Error Message

at which the error was thrown. An example of the error thrown is shown below in Figure 5.2. With this information, the user is given the necessary details to locate the position of the error within the Java code. From there, the user can take the necessary steps to correct the code.

The use of the SOOT framework allows us to implement an analysis tool captures API usage rules to detect invalid API calls. Our prototype is composed of three packages of a total of 11 classes that communicate between each other to find invalid API calls. It allows the user to define API usage rules through the use of the classes: `Method`, `Action`, `State`, `Transition` and `Configuration`. When a program is analyzed, our prototype fragments the Java byte-code into a 3 address instruction using the Jimple transformation. Then it navigates through each statement and determine the set of states for variables at each program point using the program analysis package. Finally it pinpoints any invalid API calls using the verification package and giving the user with enough information to take corrective actions towards the error.

Chapter 6

Evaluation

Using the prototype we build, we ran several tests on several programs used from tutorial sites and discussion forums. We also tested our prototype two major software called Rocket and HyperSQL.

Rocket is an enterprise data management software created and maintained by the software company Open Harbors located in St. Francis, WI [20]. Rocket allows the user to manipulate and transform any sort of data from different sources such as excel spreadsheets, CSV files, ZIP files, databases, and emails. Rocket is used by the employees of Open Harbors as well as their clients which totals about 100 current users. This software is composed of 6 sub-components, with 854 classes, 4,187 methods and over 40,000 lines of code.

HyperSQL (HSQL) is an open source software build and maintained by the HSQL Development Group [21]. HyperSQL provides a fast transactional and multithreaded relational database written entirely in Java. It offers in-memory and disk-based tables for data storage. HyperSQL is used by various software such as OpenOffice, Kepler, Mathematica, JBoss and many more. HSQL is composed of 741 classes, 10,037 methods and with over 160,000 lines of code.

To be able to work properly, Rocket and HyperSQL uses several libraries such as JDBC library, Java IO library, OpenCSV library, FTPClient library, and much more. Because those twos software generates several calls to the database and files, we focused our evaluation on the JDBC library and the Java IO library and we compared the results with the popular analysis tool FindBugs.

6.1 What is FindBugs?

FindBugs is a popular program analysis tool developed by Bill Plugh, a computer science professor at the University of Maryland Institute for Advanced Computer Studies, and David Hovemeyer, a Research and Development software engineer at Advantest Europe. This analysis tool helps finding bugs within a Java program such as null pointer exceptions, deadlocks, infinite recursive loops and bad uses of Java libraries [2]. FindBugs uses more than 200 bug patterns to detect errors within a program. It is also an open source program that provide plug-ins for major IDEs such as Eclipse, NetBeans, and inteliJ. In our evaluation, we installed an Eclipse plug-in of FindBugs to compare its results with our prototype.

6.2 Test Plan

In order to capture the full potential of our model, we developed a series of tests that can be categorized as followed:

Program Analysis
: This category tests the ability for our model to traverse the analyzed program by finding all the classes, methods, statements, and variables of the program. It will also tests its ability to build a precise call graph and program flow used during the analysis.

API Specification
: It verifies the ability of the model to properly capture API usage rules declared by the user. It also involves detecting any API method calls within the program.

Invalid API usage
: This category examines the ability of our prototype to catch all the invalid API usage based on the API specifications.

The first two categories will make sure that our model is able to properly run a Dataflow analysis of the program that is inter-procedural and context sensitive while finding all the method calls of the API being analyzed. The third category which verifies for the capability of our prototype to catch invalid API calls will tests the test cases below. Those test cases represented in Tables 6.1 and 6.2 have been put together from common JDBC and Java IO library errors reported on FindBugs and StackOverflow [15].

Case Name	Description	Expected Result
Driver not loaded	The JDBC Driver hasn't been loaded before started any database operations	Show error of invalid API usage
Statement with no connection	A statement is being created using a with no connection	Detect invalid transition from *NotConnection* to *Statement*
Result set with no statement	A result set is being created with no statement	Detect invalid transition from *StatementClosed* to *ResultSet*
Statement used after closing connection	A statement previously created with an open connection is being re-used to create a result set after the connection has been closed	Update the connection variable to *NotConnection* state and the statement variable to *StatementClosed*. It should then detect an invalid transition from *StatementClosed* to *ResultSet*.
Result set used after closing a statement	A result set previously created with a valid statement is being re-used to print data from the database after the statement has been closed	Update the statement variable to *StatementClosed* state and the result set variable to *ResultSetClosed*. It should then detect that there's no transition from *ResultSetClosed* to other states except *ResultSetClosed*
Null-Pointer Exception	A statement is being created using a null connection	Detect invalid transition from *NotConnected* to *Statement*

Table 6.1: JDBC Test Cases

Case Name	Description	Expected Result
Read file with reader closed	Trying to read a file after closing the reader	Detect invalid Transition from *NotConnected* state to *Read* state
Write file with writer closed	Trying to write to a file after closing the writer	Detect invalid transition from *NotConnected* state to *Write* state
Buffered reader used after closing file reader	A buffered reader created with a valid file reader is being re-used to read a file after the file reader has been closed	Update file reader variable to *NotConnected* state and buffered reader variable to *BufferedClosed* state. Detect invalid transition from *BufferedClosed* state to *Read* state
Buffered writer used after closing file writer	A buffered writer created with a valid file writer is being re-used to write to a file after the file writer has been closed	Update file writer variable to *NotConnected* state and buffered writer variable to *BufferClosed* state. Detect invalid transition from *BufferClosed* state to *Write* state

Table 6.2: Java I/O Test Cases

6.3 Results and Comparison with FindBugs

After testing our model on the test cases above we were able to get the results displayed in Tables 6.3, 6.4, and 6.5.

The program analysis test shows that our model was able to find most of the classes, methods and variables. This is explained by the fact that some of those classes with the

Software	Program Details		
	Classes	Methods	Variables
Rocket	154/170	322/427	189/224
HyperSQL	500/741	8,548/10037	6,245/7,937

TABLE 6.3: Test Program Analysis Results

Library	API Specifications			
	Methods	Actions	States	Transitions
JDBC	6/6	6/6	4/4	12/12
Java I/O	8/8	5/5	5/5	11/11

TABLE 6.4: Test API Specifications Results

Library	Case Name	Rocket		HyperSQL	
		Our Model	FindBugs	Our Model	Find Bugs
JDBC	Driver not Loaded	0/1	0/1	0/1	0/1
	Statement with not connection	11/11	0/11	8/9	0/9
	Result set with no Statement	11/11	0/11	7/9	0/9
	False positive for closing connection	9/11	0/11	7/9	0/9
	False positive for closing statement	9/11	0/11	7/9	0/9
	Null-pointer exception	7/11	11/11	5/9	9/9
Java IO	Read file with reader closed	11/12	12/12	10/12	12/12
	Write file with writer closed	4/5	5/5	3/5	5/5
	False positive for closing file reader	8/12	12/12	9/10	10/10
	False positive for closing file writer	3/5	5/5	3/5	5/5

TABLE 6.5: API Usage Verification Test Results

associated methods and variables extended or implemented from a parent class within a different component. Unfortunately, SOOT gets confused because the parent class and the child class are not within the same component. By having a main project composed of multiple small projects, SOOT is not able to detect cross references of classes between the small projects. It assumes that all the classes are within the project being analyzed. This is the reason why by referencing a class from another project, our prototype is not able to conduct an analysis of that referenced class.

In addition, our prototype is not heap-sensitive. In fact, it does not keep track of variables defined within objects being used in the program. It considers only variables involved in the method being called. Let's consider a class `classA` with variable `variableA` and a class `mainClass` where an instance of `classA` is being used. By analyzing `mainClass` we can explore and analyze all the methods of `classA` being called. While analyzing those methods, we can determine the states of `variableA`. Once the analysis comes back to `mainClass` we can no longer determine the states of `variableA`. Unfortunately, SOOT does not provide an easy way to make the analysis heap-sensitive.

```
Class.forName("com.mysql.jdbc.Driver");
conn = DriverManager.getConnection(DB_URL,USER,PASS);
conn.close();
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * from country");

while(rs.next()){
    ...
}
```

FIGURE 6.1: Code example with error

It requires the download and use of 3 different libraries, and 2 files produced by a program written in C. The C program was not able to produce the proper executable files to use for the analysis. Several attempts have been done to implement field sensitivity but with no success. On the API specifications test, we notice that our prototype was able to perfectly capture the user API specifications. This is due to the fact that those specifications are defined using the prototype itself. Lastly, the verification of API usage shows that our prototype is able to detect improper use of API libraries. We can see that our model is able to conduct a proper Program analysis, while taking in account the user API usage rules. It is also able to detect most of invalid API calls.

Those results prove that our model is able to detect invalid API calls by using the user API specifications on an inter-procedural context-sensitive Dataflow analysis. By running the same tests with FindBugs, we found out that our model was able isolate most of the invalid API calls that are likely to occur at run-time while FindBugs was able to detect unused variables and null pointer exceptions. Furthermore, our model detects state dependencies within the program. Considering the code snippet in Figure 6.1, our model was able to find that the statement connection was closed after closing the connection to the database.

However, our model does not capture errors outside of the API usage. In fact, it is maintaining the state of each variable in relation to the API analyzed. In other words, our model will only inspect variables that are using the analyzed API. Therefore, the model will capture most of the API invalid calls but not the other errors. We also noticed that our model runs longer than FindBugs. This is due to the fact that SOOT loads the java libraries which are necessary for the inter-procedural analysis. It therefore creates an extensive `callgraph` during the Jimple transformation.

6.4 Conclusions

The results of our testing are proven to be satisfying. Our model was able to conduct a full program analysis by finding most of the invalid API calls based on the user's input. We were able to properly implement the idea proposed in this thesis to practice with positive results. Compared with FindBugs, we were able to detect more invalid API calls. With these results, we are confident of the validity of our model.

6.5 Extensibility of Prototype

Because, our prototype forces the explicit definition of state transitions, it is easily expendable to any API transitions. It gives the ability for the user to define state transitions for the desired APIs to check. Let's consider the example of a program that uses the JDBC, FTP, and HTTP libraries. By defining the transitions for each of the libraries, we can conduct a single analysis that will detect all the invalid API calls. For that we consider our prototype to be fully expendable to most API libraries.

Chapter 7

Related Work

In relation to our model presented in this paper, there are other analysis tools that have done similar work. They are FindBugs, PRIME which is a pattern based analysis [4], GraPacc another pattern based analysis [22], CriticAL, using symbolic execution[3], and few more.

With the help of FindBugs, Grimaud and Standaert [23], design an approach similar to ours to find improper use of API libraries in Java card applets. Indeed, their approach tracks the state of each program point and checks for state correctness for all method calls using a Dataflow analysis. It first traverses the program to build a control flow graph, then it loads the appropriate bugs detectors from FindBugs for the appropriate library. An execution plan is then created in order to specify how and where the bug detector should be apply. At last, FindBugs is ran applying its bug detectors to find invalid API usage. Even though, Grimaud and Standaert's method seem efficient, it is only specific to Java card applets and it does not allow the user to add more API usage rules. The analysis is limited to the bug detectors patterns found by FindBugs. By that, API usage bugs that have not yet been reported will not be found by FindBugs, and therefore it will provide an inaccurate analysis. Our approach gives the user control over the API usage rules. It allows for the detailed specification of API rules by the user. For that, the user can immediately specify any rules that will take in account bugs that have not yet been reported to FindBugs. This makes our approach more accurate than the one from Grimaud and Standaert.

PRIME produces a type state graph pattern using code snippets found on the Internet. Even though it is able to detect invalid API calls, PRIME's accuracy is questionable. Because its API specifications are built from code snippets found on popular forums sites, PRIME's accuracy is as good as the amount of documentation found on those sites. In case there are little to no information about and API library, there is a high

chance that PRIME will miss some critical state transitions which leads to an inaccurate API specifications. In addition, PRIME does not allow the user to define the API rules to use for the analysis. The user is limited to the API specifications build by PRIME for its analysis. Therefore, PRIME's can lead to incorrect results when analyzing a program's API usage. Contrarily to PRIME, our approach allows the user to explicitly define the API specifications which lead to an accurate analysis of the program.

Similar to PRIME, GraPacc is another pattern based analysis is mainly used as a code completion tool for API usage. It provides the user with a list of suggestions for the API methods calls based on the previous API method calls in the program. That GraPacc directs the user on which API method to call next within the program. This analysis tool gathers its API specifications from a database and it allows the user to upload an API usage pattern for the analysis. One major difference between GraPacc and our approach is that GraPacc uses a set of algorithms to compute a list of suggested API methods, while we only check for a valid transition. In fact, GraPacc builds a database of API patterns based on the API libraries used within the program [24]. Those patterns are then ranked based on the pattern followed in the program. The patterns with a high rank are used to check for API usage, and suggested methods within the code. Even though our model does not support suggestions for now, our analysis is done in one step by checking for valid transitions. This operation is simple enough to have a small computation time of $O(1)$ compared to GraPacc who's ranking method that can take up to $O(n^2)$ with n being the number of patterns available.

Furthermore, CriticAL[3] is another analysis tool that provides critics and suggestions for API usages through symbolic execution. It takes in account user's input and builds a set of rules for the verification. What makes our code different from CriticAL is the technique used. CriticAL, using symbolic execution, provides an analysis based on the execution path of the code. It means that some part of the program will not be evaluated as the execution does not reach them. For that, CriticAL is likely to omit some missing calls as part of the code has not been analyzed. Our model explores all the possibilities of execution within the code, and therefore catching all the API invalid calls.

In addition, The SLAM Project [25] has been developed by Ball and Rajamani with the same objective as our approach: verify API usage. This project is a program that checks proper use of API in programs written in C language by focusing on Windows XP drivers through the use of the Specification Language for Interface Checking (SLIC). Its analysis is done in three phases. The first one transforms the C program into a Boolean program, using the transformation tool C2BP. The second phase consists of applying the reachability tool BEBOP to perform a reachability analysis. This analysis will focus on find paths that leads to ERROR state within the Boolean program. The last phase

is to use the tool NEWTON to check for the validity of a path with the C program. The SLAM project's analysis is based on a reachability analysis. It explores reachable paths to ERROR and tests their feasibility. However, the SLAM project could be very slow at times. It takes minutes to half hour to analyze about 10,000 lines. This is because the reachability analysis could provide an exponential number of states in the worst case.

Monperrus and Mezini [26] introduce a method called Detector of Missing Method Calls (DMMC) to detect missing method calls for API usage. It is based on a S-score that determine deviant type-usage of APIs instances. This analysis is based on a usage majority rule. It constructs a list of methods for each instance of the API. It then compares the sequence of methods for each type and it divides them into two groups: "exactly-similar and almost-similar" type-usage. A score called S-score is determine based on the total of methods in each group. The S-score is low when the group "exactly-similar" contains more methods that the group "almost-similar". It is high otherwise. That score determine the likeliness of having bugs while using an API. Through the DMMC, Monperrus and Mezini are able to find violation of API best practices, aging software, and dead code. We believe that this approach could alert the developer for a possible bug for every instance of a API when the type-usage is slightly different. For example, let's consider a program that instantiate an API within 5 different classes. Two of those instances have exactly the same sequence of calls while the other three have each a additional method. The DMMC is likely to provide a low S-score to these three type-usages because of the additional method. The developer is then alerted of a possible bug when there is no bug. This is why we believe that this technique is effective in programs that use API calls uniformly.

In the works of defining API specifications, Ammons et al. [1] in *Mining Specification* introduced a technique to gather API specifications from programs. This technique produces dependency flows from traces of API calls within a program. It then uses those dependency flows to build scenarios that are consolidated into API specifications. Unfortunately, this technique relies on two assumptions that could lead to the consolidation of incorrect API specifications.

Correct use of API method calls in the analyzed program

This assumption could result in incorrect API specifications in the case that the calls are incorrect. It could also lead into incomplete API specifications due to the fact that not all the methods are used within the program.

"Common behavior is often correct behavior"

This second assumption might also produce incomplete API specifications as the algorithm used to consolidate scenarios into specifications omits API method calls that are not commonly used.

Further into mining API specifications, Acharya et al. [6] mines specifications using the source code of the API. It therefore guarantees the build of complete and accurate API specifications compared to Ammons et al [1]. It also uses traces to build scenarios that when consolidated generate Frequent Closed Partial Orders (FCPO) [6]. Those FCPO are then put together using a *Mine-Verify* algorithm to produce API specifications. Unfortunately Acharya et al's technique is not flow sensitive, which might cause a loss of accuracy during the mining process.

Contrary to Ammons et al. [1] and Acharya et al. [6], Lo and Maoz [27] base their API mining tool on object identity and lifetime. In other words, their technique considers the object lifetime and dependencies during the program execution. It extracts API specifications from classes that are instantiated "on-the-fly" and from classes that are used for a short period of time during the program execution. This technique is useful when analyzing a game program such as Space Invader where some events occur at any point in time at the request of the player. An example would be for the player to request the monster to fire a laser beam at an object. Lo and Maoz, use a Live Sequence Chart (LSC) [28] to link scenarios and events. The LSCs are then combine to produce API specifications after satisfying the minimum requirements for confidence and support threshold [29]. Even though this technique dynamically produces API specifications and considers object lifetime, it is highly dependent on the accuracy of the API traces used to build the LSC.

In the article "Statically Checking API Protocol Conformance with Mined Multi-Object Specifications", Pradel et al. [30] combine a dynamic specification mining to a static code verification analysis to detect illegal use of API calls. They use a mining tool to dynamically build API specifications that are used during a static analysis of a program. Pradel et al.'s technique uses a state relationship between object interactions to determine proper usage of and API method. Even though this technique is similar to ours, we believe that the accuracy of the API specification depends on the amount of example found.

Chapter 8

Conclusion and Future Work

In our experiment, we proposed a technique to analyze API usage in programs through temporal states. We were able to detect invalid API calls through a Dataflow context sensitive analysis. Our prototype, built on top of the SOOT framework, was able to show satisfying results. It was able to detect bugs from programs which other analysis tool were not able to detect such as FindBugs. Even though we are confident that our solution contributes to finding hidden bugs in programs, there is room for future work. One area of future work is the creation of a database or a source file with state transitions already defined. This make the analysis faster because there is no need to manually defined state transitions for the analyzed API. Another area of future work is the support for more APIs. In fact, each API has different usage rules. By implementing those rules, our prototype could analyze a wider range of APIs, and therefore analyze all API calls in a program. Furthermore, the idea of creating a plugin that will work for most IDEs could be explored, allowing developers to directly integrate it in their development environment. Even though our prototype is still in the early stages of its development, we are confident that this thesis introduces a technique that will improve program analyses through the explicit definition of state transitions.

Bibliography

[1] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, January 2002. ISSN 0362-1340. doi: 10.1145/565816.503275. URL http://doi.acm.org/10.1145/565816.503275.

[2] Source Forge. Findbugs, November 2013. URL http://www.findbugs.sourceforge.net.

[3] Chandan R. Rupakheti. *A Critic for API Client Code using Symbolic Execution*. PhD thesis, Clarkson University, May 2012.

[4] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. *SIGPLAN Not.*, 47(10):997–1016, October 2012. ISSN 0362-1340. doi: 10.1145/2398857.2384689. URL http://doi.acm.org/10.1145/2398857.2384689.

[5] Raymond P. L. Buse and Westley Weimer. Synthesizing api usage examples. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 782–792, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337316.

[6] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining api patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 25–34, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4. doi: 10.1145/1287624.1287630. URL http://doi.acm.org/10.1145/1287624.1287630.

[7] Tao Xie and Jian Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 54–57, New York, NY, USA, 2006. ACM. ISBN 1-59593-397-2. doi: 10.1145/1137983.1137997. URL http://doi.acm.org/10.1145/1137983.1137997.

[8] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 174–184, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-734-6. doi: 10.1145/1273463.1273487. URL http://doi.acm.org/10.1145/1273463.1273487.

[9] Gias Uddin, Barthelemy Dagenais, and Martin P. Robillard. Analyzing temporal api usage patterns. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 456–459, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4577-1638-6. doi: 10.1109/ASE.2011.6100098. URL http://dx.doi.org/10.1109/ASE.2011.6100098.

[10] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Spinger, 1999, 2005.

[11] Byeong-Mo Chang and Jangwu Jo. Granularity of constraint-based analysis for java. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, pages 94–102, New York, NY, USA, 2001. ACM. ISBN 1-58113-388-X. doi: 10.1145/773184.773195. URL http://doi.acm.org/10.1145/773184.773195.

[12] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the Occasion of His Retirement from His Professorship at the University of Kiel)*, pages 114–136, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66624-9. URL http://dl.acm.org/citation.cfm?id=646005.673740.

[13] Christopher Doble, Colin J. Fidge, and Diane Corney. Data flow analysis of embedded program expressions. In *Proceedings of the Tenth Australasian Information Security Conference - Volume 125*, AISC '12, pages 71–82, Darlinghurst, Australia, Australia, 2012. Australian Computer Society, Inc. ISBN 978-1-921770-06-7. URL http://dl.acm.org/citation.cfm?id=2512113.2512122.

[14] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica - ACTA*, 7(3):305–317, 1977.

[15] Stack Exchange Inc. Stackoverflow, November 2014. URL http://stackoverflow.com.

[16] Sable Research Group at McGill University. Soot, January 2012. URL http://www.sable.mcgill.ca/soot.

[17] Rohan Padhye and Uday P. Khedker. Interprocedural data flow analysis in soot using value contexts. In *Proceedings of the 2Nd ACM SIGPLAN International*

Workshop on State Of the Art in Java Program Analysis, SOAP '13, pages 31–36, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2201-0. doi: 10.1145/2487568.2487569. URL http://doi.acm.org/10.1145/2487568.2487569.

[18] Arni Einarsson and Janus Dam Nielson. *A Survivor's Guide to Java Program Analysis with Soot*, July 2008.

[19] Patrick Lam. *Using the SOOT flow Analysis Framework*, March 2000.

[20] Open Harbors. Open harbors, November 2012. URL https://openharbors.com/.

[21] HSQL Development Group. Hypersql, February 2014. URL http://hsqldb.org/.

[22] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 69–79, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337232.

[23] Gilles Grimaud and François-Xavier Standaert. *Smart Card Research and Advanced Applications*, volume 5189 of *0302-9743*. Springer Berlin Heidelberg, September 2008.

[24] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. Mining succinct and high-coverage api usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 319–328, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. URL http://dl.acm.org/citation.cfm?id=2487085.2487146.

[25] *The SLAM project: debugging system software via static analysis*, January 2002. ACM New York, NY, USA.

[26] Martin Monperrus and Mira Mezini. Detecting missing method calls as violations of the majority rule. *ACM Trans. Softw. Eng. Methodol.*, 22(1):7:1–7:25, March 2013. ISSN 1049-331X. doi: 10.1145/2430536.2430541. URL http://doi.acm.org/10.1145/2430536.2430541.

[27] David Lo and Shahar Maoz. Specification mining of symbolic scenario-based models. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '08, pages 29–35, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-382-2. doi: 10.1145/1512475.1512482. URL http://doi.acm.org/10.1145/1512475.1512482.

[28] Werner Damm and David Harel. Breathing life into message sequence charts. *Formal Methods in System Design*, 19:45–80, 2001.

[29] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2 edition, 2006.

[30] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking api protocol conformance with mined multi-object specifications. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 925–935, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337332.